Elsevier Editorial System(tm) for Expert Systems With Applications

Manuscript Draft

Manuscript Number:

Title: Diagnosis Using a First-Order Stochastic Language That Learns

Article Type: Full Length Article

Section/Category:

Keywords: Stochastic modeling, loopy belief propagation, parameter estimation.

Corresponding Author: Chayan Chakrabarti,

Corresponding Author's Institution: University of California, Irvine

First Author: Chayan Chakrabarti

Order of Authors: Chayan Chakrabarti; Rosham Rammohan; George F Luger

Manuscript Region of Origin:

Abstract:

Diagnosis Using a First-Order Stochastic Language That Learns

Chayan Chakrabarti, Roshan Rammohan, and George F. Luger

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
{cc, roshan, luger} @cs.unm.edu

Abstract

We have created a diagnostic/prognostic software tool for the analysis of complex systems, such as monitoring the "running health" of helicopter rotor systems. Although our software is not yet deployed for real-time in-flight diagnosis, we have successfully analyzed the data sets of actual helicopter rotor failures supplied to us by the US Navy. In this paper, we discuss both critical techniques supporting the design of our stochastic diagnostic system as well as issues related to its full deployment. We also present four examples of its use.

Our diagnostic system, called DBAYES, is composed of a logic-based, first-order, and Turing-complete set of software tools for stochastic modeling. We use this language for modeling time-series data supplied by sensors on mechanical systems. The inference scheme for these software tools is based on a variant of Pearl's loopy belief propagation algorithm (Pearl, 1988). Our language contains variables that can capture general classes of situations, events, and relationships. A Turing-complete language is able to reason about potentially infinite classes and situations, similar to the analysis of dynamic Bayesian networks. Since the inference algorithm is based on a variant of loopy belief propagation, the language includes expectation maximization type learning of parameters in the modeled domain. In this paper we briefly present the theoretical foundations for our first-order stochastic language and then demonstrate time-series modeling and learning in the context of fault diagnosis.

1. Introduction

The paper presents the results of our efforts in the analysis and diagnosis of complex situations, such as those found in data from sensors attached to various components of helicopter rotor systems. We have been working for the past four years in the application of a first-order stochastic modeling language for this and similar domains. We feel that a first-order and Turing-complete stochastic system is appropriate for these tasks since it supports the creation of general variable based rule relationships (the expressive power of the first-order predicate calculus) as well as supports (with fully implemented recursion) time-series analysis. This paper describes these software tools and the methodology used to address the real time diagnosis of the time-series data of the helicopter rotor systems.

Our research began with NSF support to the third author for developing tools for diagnosis using stochastic approaches. The result of this research was the creation (in OCAML) of a set of tools for diagnosis and prognosis (Pless and Luger, 2001, 2003). These stochastic software tools were both first-order and Turing complete. Subsequent to that effort the third author was also awarded SBIR and STTR contracts from the US Navy (through a small software company in Albuquerque, NM, Management Sciences, Inc.) to develop a Java based software toolkit for performing stochastic modeling. As part of this contract, the US Navy supplied to the authors real-time sensor data from helicopter rotor systems. The application of our toolkit to this data, along with several other examples of diagnosis/prognosis is the theme of this paper.

The ideal next step for our current software will be to embed it in the control systems that monitor complex devices. But this will require further development, including the application of our algorithms to more data sets and creating the appropriate software for integrating these algorithms into existing flight control systems. Our concluding section presents these issues further.

Section 2 of this paper gives a brief overview of the theoretical issues supporting the development of our logic-based stochastic modeling language. In Section 3, we present a direct application of our software to time-series data for the purpose of fault diagnosis. We show that the fully recursive nature of our language is ideal for supporting variants of hidden Markov models doing time-series analysis.

Because our inference scheme is based on a variant of Pearl's loopy belief propagation (Pearl, 1988) it is also ideally suited for expectation maximization type learning. We demonstrate this in fitting parameters to components of a stochastic model. The learning of model components is described in Section 4.

Finally, in Section 5 we present our thoughts on the research/application issues that remain in this project. The current Java version of our software is available from the authors.

2. DBAYES: A logic-based stochastic modeling language

In this section we briefly describe the formal foundations of our logic-based stochastic modeling language. We have extended the Bayesian logic programming approach of Kersting and De Raedt (2000) and have specialized the Kersting and De Raedt representational formalism by suggesting that product distributions are an effective combining rule for Horn clause heads. We have also extended the Kersting and De Raedt language by adding learnable distributions. To implement learning, we use a refinement of Pearl's (1998) loopy belief propagation algorithm for inference. We have built a message passing and cycling - thus the term "loopy" - algorithm based on expectation maximization or EM (Dempster et al., 1977) for estimating the values of parameters of models built in our system. Further details of this learning component are presented in Section 4. We have also added additional utilities to our logic language including second order unification and equality predicates.

A number of researchers have proposed logic-based representations for stochastic modeling. These first-order extensions to Bayesian Networks include probabilistic logic programs (Ngo and Haddawy, 1997) and relational probabilistic models (Koller and Pfeffer, 1998; Getoor et al., 1999). The paper by Kersting and De Raedt (2000) contains a survey of these logic-based approaches. Another approach to the representation problem for stochastic inference is the extension of the usual propositional nodes for Bayesian inference to the more general language of first-order logic. Several researchers (Kersting and De Raedt, 2000; Ngo and Haddawy, 1997; Ng and Subrahmanian, 1992) have proposed forms of first-order logic for the representation of probabilistic systems.

Kersting and De Raedt (2000) associate first-order rules with uncertainty parameters as the basis for creating Bayesian networks as well as more complex models. In their paper "Bayesian Logic Programs", Kersting and De Raedt extract a kernel for developing probabilistic logic programs. They replace Horn clauses with conditional probability formulas. For example, instead of saying that `x` is implied by `y` and `z`, that is, `x <- y, z` they write that `x` is conditioned on `y` and `z`, or, `x | y, z`. They then annotate these conditional expressions with the appropriate probability distributions.

Our research also follows Kersting and De Raedt (2000) as to the basic representation structure of the language. A sentence in the language is of the form:

$$\texttt{head} \mid \texttt{body}_1, \texttt{body}_2, .., \texttt{body}_n = [p_1, p_2, \ldots, p_m]$$

The size of the conditional probability table (m) at the end of the sentence is equal to the arity (number of states) of the head times the product of the arities of the terms in the body. The probabilities are naturally indexed over the states of the head and the clauses in the body, but are shown here with a single index for simplicity. For example, suppose `x` is a predicate that is valued over `{red,green,blue}` and `y` is boolean. `P(x|y)` is defined by the sentence

```
X | y =[[0.1,0.2,0.7],[0.3,0.3,0.4]]
```

here shown with the structure over the states of `x` and `y`. Terms (such as `x` and `y`) can be full predicates with structure and contain PROLOG style variables. For example, the sentence `a(X) = [0.5,0.5]` indicates that `a` is (universally) equally likely to have either one of two values.

If we want a query to be able to unify with more than one rule head, some form of combining function is required. Kersting and De Raedt (2000) allow for general combining functions, while the Loopy Logic language restricts this combining function to one that is simple, useful, and works well with the selected inference algorithm. Our choice for combining sentences is the product distribution. For example, suppose there are two simple rules (facts) about some Boolean predicate `a`, and one says that `a` is `true` with probability 0.4, the other says it is `true` with probability 0.7. The resulting probability for `a` is proportional to the product of the two. Thus, `a` is `true` proportional to 0.4 * 0.7 and `a` is `false` proportional to 0.6 * 0.3. Normalizing, `a` is `true` with probability of about 0.61. Thus the overall distribution defined by a database in the language is the normalized product of the distributions defined for all of its sentences.

One advantage of using this product rule for defining the resulting distribution is that observations and probabilistic rules are now handled uniformly. An observation is represented by a simple fact with a probability of 1.0 for the variable to take the observed value. Thus a fact is simply a Horn clause with no body and a singular probability distribution, that is, all the state probabilities are zero except for a single state.

Our software also supports Boolean equality predicates. These are denoted by angle brackets <>. For example, if the predicate `a(n)` is defined over the domain `{red, green,blue}` then `<a(n) = green>` is a variable over `{true,false}` with the obvious distribution. That is, the predicate is `true` with the same probability that `a(n)`is `green` and is `false` otherwise.

The next section demonstrates the use of our software in diagnosing faults, where sensor data is captured across ordered slices of time. Then the following section address issues of parameter fitting with EM-type learning.

3. Inference in Loopy Logic

In Kersting and De Raedt's work, inference proceeds by constructing an SLD (Selection rule, Linear resolution, Definite clauses) tree (a selective literal resolution system for definite clauses) and then converting it into a Bayesian Network. Loopy Logic follows a similar path, but instead converts the SLD tree to a Markov field. The advantage of this approach is that the product distributions that arise from goals that unify with multiple heads can be handled in a completely natural way. The basic idea is that random variable nodes are generated as goals are found. Cluster nodes are created as goals are unified with rules. In a logic program representing a Bayesian Network, the head of a statement corresponds to a child node, while the clauses in the body correspond to the node's parents as shown in Figure 1. To construct a Markov field, Loopy Logic adds a cluster node between the child and its parents. If more than one rule unifies with the rule head, then the variable node is connected to more than one cluster node.

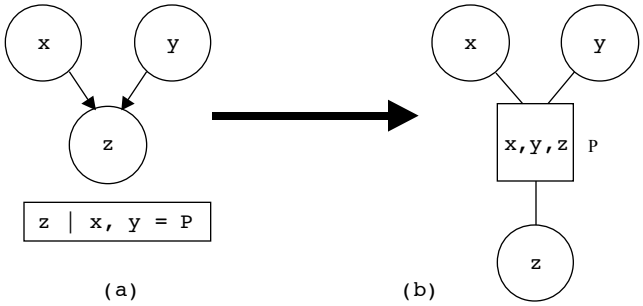

Figure 1. The transition of a Bayesian network into an equivalent Markov random field.

As a result of the addition of the cluster nodes, the graphs that are generated for inference are bipartite as shown in Figure 3.1(b). There are two kinds of nodes in these graphs, the variable and the cluster nodes. The variable nodes hold distributions for the random variables they define. The cluster nodes contain joint distributions over the variables to which they are linked. Messages between nodes are initially set randomly. On update, the message from variable node V to cluster node C is the normalized product of all the messages incoming to V other than the message from C. In the other direction, the message from a cluster node C to a variable node V is the product of the conditional probability table (local potential) at C and all the messages to C except the message from V. This product is marginalized over the variable in V before being sent to V. This process, starting from random messages, and iterating until convergence, has been found to be effective for stochastic inference (Murphy et al. 1999).
The algorithm works by starting from a query (or possibly a set of queries) and generating the variable nodes that are needed. Each query is matched against all unifying heads in the database. All the ground facts must also be included in the network. The resulting bodies are then converted to new goals in the search. Loopy Logic is limited in that the goals produced by this search must be ground terms, the "facts" of the modeled domain, where we set the probability of the variable to one. Kersting and De Raedt (2000) place a range restriction on variables in terms: a variable may appear in the head of a rule only if it also appears in the body. As a result of this requirement, all facts entailed from the database are ground. By contrast, Loopy Logic requires that all entailed goals be ground. We have found that this requirement makes for better construction of useful models.

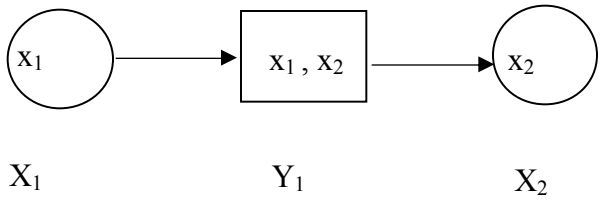

Figure 2. Message passing in Loopy Logic.

The message passed from a variable node to a cluster node is the normalized product of all the messages incoming to the variable node other than the message from the cluster node itself. For example, in Figure 2, the message from variable node X_1 to cluster node Y_1 is the normalized product of incoming messages, say from many cluster nodes, Y_1, Y_2, etc to X_1. In the other direction, the message from a cluster node Y_1 to a variable node is the product of the conditional probability table (local potential) at the cluster node and all the messages incoming to the cluster node except the message from the variable node. Before passing to the variable node, the message is marginalized based on the variable. For example, if the conditional probability table at cluster node Y_1 is P_1, then the message from cluster node Y_1 to variable node X_2 is the normalized product of P_1 and the message from other variables nodes X_1, X_3, etc (except X_2) to Y_1. The product table is marginalized based on X_2 before passing to X_2.

4. Fault diagnosis using variants of hidden Markov models

We now consider the application of our stochastic modeling software to fault diagnosis in complex mechanical systems, such as in the rotor assemblage of Navy helicopters. Before discussing the Navy data, we present a simple example showing how to construct a hidden Markov model (HMM) in our declarative Bayesian logic.

Example One: A Simple Hidden Markov Model

In this example, there are two states (x, y). The system can start in either one, and at each time step, cycle to itself or transition to the other state. The probability of these events is a learnable distribution. In both states, the system can output one of two symbols (a, b). The conditional distribution for these emissions is also represented in this model by an adjustable distribution.

```
state <- {x,y}.
emit <- {a,b}.
state(s(N)) | state(N) = State.
emit(N) | state(N) = Emit.
```

The hidden Markov model works as follows. Each state is represented with an integer that is zero or the successor of another integer. An integer *shorthand* is implemented in this system, i.e., `2` is shorthand for `s(s(0))`. In the model, each state is conditioned on the previous state with the learnable distribution `State`. Each state emits its output with the learnable distribution `Emit`.

Strictly speaking, because of the representational flexibility of our stochastic logic language, the previous four lines of code are sufficient to specify an HMM. The next five lines are included to demonstrate the utility of several of our other extensions. Note, for example, the definition of the `and` predicate:

```
observed,o,and <- {true,false}.
and(X,Y)|X,Y = [true,false,false,false].
o([],N) = true.
o([H|T],N) = and(<emit(N)=H>, (T,s(N))).
observed(L) = o(L,0).
```

Without these last five lines, one must specify an observed sequence by including in the database a separate fact for each emission that is seen. That is, one must state `emit(0) = a, emit(1) = b, emit(2) = b` and so on. With the additional five lines, three observations can be included with the predicate `observed([a,b,b])`.

A product of HMMs is expressed by adding a new predicate to indicate the states of a second HMM. This new HMM can be coupled to the existing one through a product distribution by using the same `emit` predicate.

Here is an example of a second HMM with three states:

```
state2 <- {z,q,w}.
state2(s(N))|state2(N)= State2.
emit(N) | state2(N) = Emit2.
```

Note that the final line uses the previous `emit` predicate which creates the product distribution. As a final comment, our logic-based stochastic language offers far more generality than is required to represent simple HMMs; the next example shows an extension of this approach.

Example Two: Data analysis of helicopter rotor systems using an auto-regressive hidden Markov model

In the previous example, we presented a simple HMM problem and its solution in the OCAML software representation. In the present example we make a much more complex analysis of prognosis in a complex environment. The time-series data was obtained from sensors monitoring helicopter rotors for the United States Navy. The task was to construct a quantitative model of the whole process and use it to predict faults. Various techniques were investigated for preprocessing the data. Methods of modeling the system included simple correlative classification as well as hidden Markov models (Chakrabarti et al., 2005). We also used our Java software with full recursion to replace the simple (preset) iteration of the OCAML HMM solution of *Example One*.

The data sets were collected over a period of time during which a fault was seeded in the mechanical process. For example, missing teeth in a gear or a crack in the drive shaft. The sensors were typically thermocouples and vibration meters that are continuous and analog devices. The data was sampled from the readings and made available in digital format. Figures 3 and 4 show such a data sample.

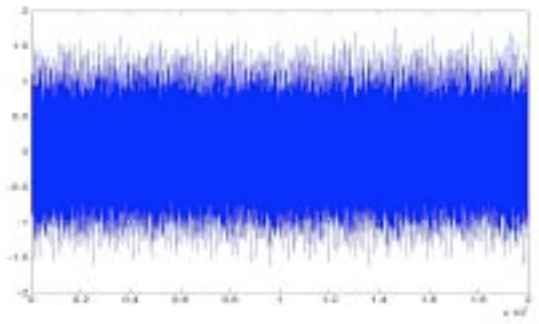

Figure 3. Raw time-series data obtained directly from mechanical processes

As can be seen in Figure 3, the raw data is intractable, noisy and unsuitable for any sort of mathematical or logical analysis. In order to get a better understanding on the nature of the data, it proved necessary to look at its frequency characteristics. The frequency spectrum of the data was calculated using the fast Fourier transform algorithm. The data in this form proved more tractable as is shown in Figure 5.

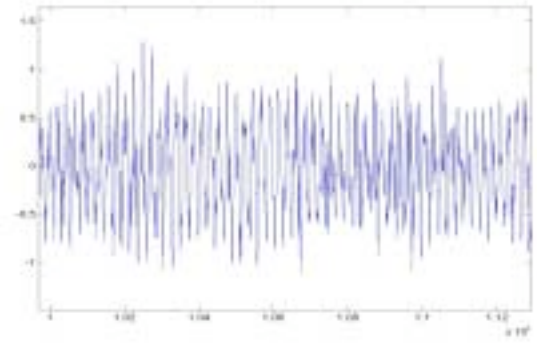

Figure 4. A zoomed in view of the time series data presented in Figure 3.

To get rid of artifacts due to noise in the frequency domain representation of the data and to consolidate information over time we computed the mean of several such windows. These processed datasets were considered observations relevant to the consequent modeling process.

The mathematical correlation between observations was used as a metric of distance. Using this metric, correlation plots were computed between half the observations that were chosen as training data. A significant and steep drop in correlation was noticed at samples bunched around a particular point in time. This point was around two thirds of the total observation time away from the first sample. Assuming that the center point of this lack of correlation was the point that the fault characteristics peaked, the time-line was split into three regions: Safe, Unsafe and Faulted.

Using these sets of correlation plots as our "learned" model about the data and fault process, the other half of the data, the test set, was correlated with the training dataset. The best fit of these new curves to the training correlation curves were computed using the Least Mean Square

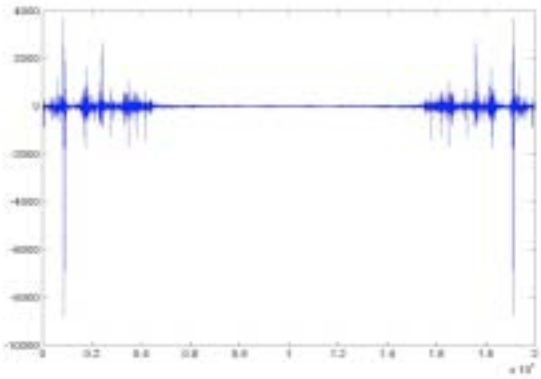

Figure 5. A "frequency domain" representation of the data computed using the Fast Fourier Transform.

metric. With this method the test data was successfully classified as Safe, Unsafe or Faulty.

Dynamic Bayesian networks *(DBNs) (Dagum et al., 1992)* can be used as a tool to model dynamic systems. More expressive than hidden Markov models (HMM) and Kalman filter Models (KFM), they can be used to represent other stochastic graphical models in Artificial Intelligence and Machine Learning.

For our model, in order to build a more robust, versatile and generic model than the above correlation-classification technique, we decided to explore the use of variants of the hidden Markov model (HMM). The auto regressive hidden Markov model (AR-HMM) (Juang, 1984) proved suitable for this purpose. The AR-HMM incorporates a causality link between consequent observations in time rather than just between states and state-observation pairs. Computationally, it provides an additional path of inference from observation of hidden state. Figure 4 shows the causality between states and observations at two consecutive instances of time (t and t-1).

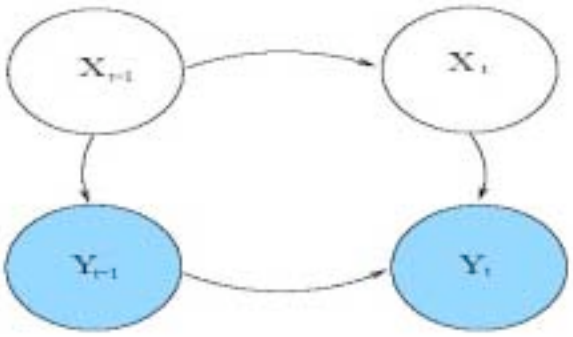

Figure 6. An Auto-regressive HMM where X_t is the state at time t, Y_t the observation of an emit value at time t. The arrows denote the causal relationships

The blank circles, labeled X are the hidden states of the system that could be one of {Safe, Unsafe, Faulted}. The shaded circles labeled Y are the observations. Before we apply the algorithm to real time data we evaluate the distribution P (u_j |X) of expected frequency signatures corresponding to the states from a state-labeled dataset. Note that U = u_1, u_2 ... u_k is the set of observations that have been recorded while training the system. Say for example, if u_1 through u_k were observed when the system gradually went from safe to faulty we would expect P(u_1 | X = safe) to be much higher than P(u_k | X = safe). See Figure 5 for a graphical representation of this probability.

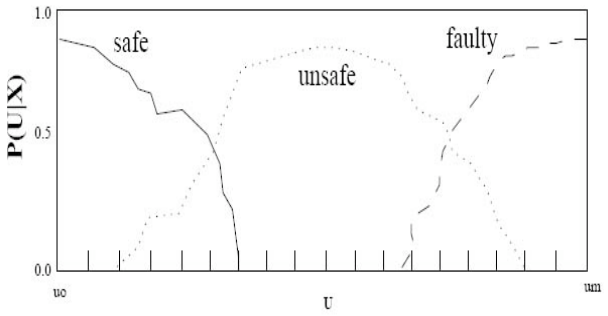

Figure 5 Probability distributions for safe, unsafe and faulty states.

The causal relationships in the AR-HMM are represented as probability distributions governed by the following equations.

$$P(Y_t = y_t \mid X_t = i, Y_{t-1} = y_{t-1}) = P(Y_t = y_t \mid X_t = i) * P(Y_t = y_t \mid Y_{t-1} = y_{t-1}) \quad (1)$$

In this design, the probability of an observation given a state is the probability of observing the discrete prior that is closest to the current observation, penalized by the distance between the current observation and the prior.

$$P(Y_t = y_t \mid X_t = i) = \max(\text{abs}(\text{corrcoef}(y_t, u_j))) * P(u_t \mid X_t = i) \quad (2)$$

Further, the probability of an observation at time t given another particular observation at time t-1 is the probability of the most similar transition among the priors penalized by the distance between the current observation and the observation of the previous time step.

$$P(Y_t = y_t \mid Y_{t-1} = y_{t-1}) = \text{abs}(\text{corrcoef}(y_t, y_{t-1})) * ((\#\text{ of } u_{t-1} \text{ to } u_t \text{ transitions}) / (\#\text{ of } u_{t-1} \text{ observations}))$$
where, $u_t = \text{argmax}_{uj}(\text{abs}(\text{corrcoef}(y_t, u_j)))$ (3)

Note that y_t is a continuous variable and potentially infinite in range but we limit it to a tractable set of finite signatures, U, by replacing it by the u_j with which it correlates best.

The relationship governing the learnable distributions is expressed as follows:

```
x <- {safe, unsafe, faulty}.
y(s(N)) | x(s(N)) = LD1.
y(s(N)) | y(N)) = LD2.
```

Preprocessing the data and computing the correlation coefficients off-line, we tested the above technique on a training set of a single seeded fault occurrence taking the system from safe to faulty. Although individual predictions per time slice matched the expected results only 80% of the time, when the predicted states were smoothed over a period of neighboring time samples, the system predicted states of the faulting system with close to 100% accuracy.

5. Learning Using Loopy Logic

In this section we demonstrate how parameter learning can be used in the context of the AR-HMM. Basically, learning is achieved by adding learnable distributions to Kersting and De Raedt's language (Pless and Luger, 2002; Pless 2003). The learning message passing algorithm is based on the concept of Expectation Maximization (EM) to estimate the learned parameters in the general case of models built in the system (Chakrabarti, 2005).

The expectation maximization (EM) algorithm was first discussed by Dempster, Laird, and Rubin (1977). This algorithm estimates learning parameters iteratively, starting with an initial guess. Each iteration of the algorithm consists of an expectation step (E step) and a maximization step (M step). In the expectation step, the distributions for the unobserved variables are based on their known value and the current estimate of the unknown parameters. The maximization step re-estimates the parameters. These two steps continue until they reach their maximum likelihood with the assumption that the distribution found in the expectation step is correct. As shown by Dempster, et al. (1977), each EM iteration increases this likelihood, unless some local maximum has already been reached.

Example Three: Parameter fitting using expectation maximization.

We return again to the OCAML representation for a simple example of parameter fitting or learning. The representational form for a statement indicating a learnable distribution is `a(X) = A`. The "`A`" indicates that the distribution for `a(X)` is to be fitted. The data over which the learning takes place is obtained from the facts and rules presented in the database itself. To specify an observation, the user adds a fact (or rule relation) to the database in which the variable `X` is bound. For example, suppose, for the rule defined above, the set of five observations (the bindings for `X`) are added to produce the database:

```
a(X)=A.
a(d1)=true.
a(d2)=false.
a(d3)=false.
a(d4)=true.
a(d5)=true.
```

In this case there is a single learnable distribution and five completely observed data points. The resulting distribution for `a` will be true 60% of the time and false 40% of the time. In this case the variables at each data point are completely determined.

In general, this is not necessarily required, since there may be learnable distributions for which there are no direct observations. But a distribution can be inferred in the other cases and used to estimate the value of the adjustable parameter. In essence, this provides the basis for an expectation maximization (Mayraz and Hinton 2000) style algorithm for simultaneously inferring distributions and estimating their learnable parameters. Learning can also be applied to conditional probability tables, not just to variables with simple prior distributions. Furthermore, learnable distributions can be parameterized with variables just as any other logic term. For example, one might have a rule:

```
(rain(X,City)|season(X,City) = R(City))
```

This rule indicates that the probability distribution for rain depends on the season and varies by city.

Example 4: Learning in the context of a life-support simulation.

Next we demonstrate learning in a space station simulation that mode a small part of an advanced life support system. The scenario involves the interaction between the power sub-system and the life support system on a remote base station. The power supply is dependent on an unknown external force and fluctuates. Life support has a number of states; {normal, stressed, critical}, that depend on power availability, demand, activity and location.

The simulation assumes one astronaut. The consumption of life support resources is a function of the astronaut's exertion level and location. Our goal is to learn the model and predict the state of the life support system. Given that life support is dependent on power and consumption, we have a learnable distribution, where N is the time step and LS is the learnable distribution:

```
life support(N) | power(N), consumption(N) =
LS.
```

The state of power can be monitored from voltage output, which can be in either of five states from very high to very low, `{vh,vmh,vm,vml,vl}`. We learn the distribution, LS by first watching emission from life support that will raise alerts, `{ok, warning, danger}` . At some point life support emissions may end, but we still need to know the state of the life support system. We can do this using the learnt distribution, `LS` .

```
consumption(N) | person_activity(N),
person_location(N)= [...].
life_support <- {normal,stressed,critical}.
ls_emit <- {ok,warning,danger}.
power <- {high,medium,low}.
power_emit <- {vh,vmh,vm,vml,vl}.
person_activity <- {sleep,normal,hi_exert}.
person_location <- {in, out}.
consumption <- {low,med,high}.
```

```
consumption(N) | person_activity(N),
person_location(N)=
[[[0.7,0.2,0.1], [0.3,0.5,0.2]],
      [[0.2,0.5,0.3],
[0.6,0.2,0.2]], [[0.2,0.5,0.3],
      [0.1,0.2,0.7]]].
life_support(N)|power(N),consumption(N)=LS.
life_support(N) | ls_emit(N) =
       [[0.7,0.2,0.1], [0.2,0.6,0.2],
       [0.1,0.2,0.7]].
power(N) | power_emit(N) =
[[0.7,0.2,0.1], [0.6,0.3,0.1],
       [0.2,0.6,0.2],
[0.1,0.3,0.6],[0.1,0.2,0.7]].
```

Here are some observations from the life support system:

```
ls_emit(1)=danger
ls_emit(2)=danger
ls_emit(3)=danger
ls_emit(4)=warning
ls_emit(5)=ok
ls_emit(6)=ok
ls_emit(7)=ok
ls_emit(8)=ok
ls_emit(9)=ok
ls_emit(10)=warning
power_emit(1)=vml
power_emit(2)=vml
power_emit(3)=vm
power_emit(4)=vmh
power_emit(5)=vmh
power_emit(6)=vh
power_emit(7)=vh
power_emit(8)=vh
power_emit(9)=vh
power_emit(10)=vh
power_emit(11)=vmh
power_emit(12)=vh
power_emit(13)=vh
power_emit(14)=vh

person_activity(1)=hi exert
person_activity(2)=hi exert
person_activity(3)=normal
person_activity(4)=normal
person_activity(5)=normal
person_activity(6)=sleep
person_activity(7)=sleep
person_activity(8)=sleep
person_activity(9)=normal
person_activity(10)=hi exert
person_activity(11)=hi exert
person_activity(12)=hi exert
person_activity(13)=hi exert
person_activity(14)=hi exert
person_location(1)=out
person_location(2)=out
person_location(3)=in
person_location(4)=in
person_location(5)=in
person_location(6)=in
person_location(7)=in
person_location(8)=in
person_location(9)=in
person_location(10)=out
person_location(11)=out
person_location(12)=out
person_location(13)=out
person_location(14)=out
```

We begin the simulation at `time` = 1 with life support in critical condition, power supply low, astronaut outside and in a state of high exertion. The power supply stabilizes around `time` = 6, and at the same time the astronaut goes to sleep. He later wakes up, begins high exertion activity and ventures outside. The power remains stable, except for a slight dip at `time` = 11. The life support emissions end at `time` = 10. Thereafter, the state of the system must be determined from the learnt distribution, `LS`. Table 1 shows the likelihood of states at each time step. The system determines that the state of life support after time step 10, when the astronaut is outside and exhibiting high exertion, is more likely to be in state `{stressed}`. This seems a logical inference because when the astronaut was in high exertion and the power level was low, the state of life support was `{critical}`. The high amount of exertion has likely put the life support system in a stressed state, but since power output is full, it is not reaching a critical state. Also note that at `time` = 11, when the power output dipped slightly, the likelihood of being in state critical was at its highest level since `time` = 3.

Life support system states:

Time	Normal	Stressed	Critical
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0.77	0.23
5	0.74	0.14	0.12
6	0.92	0.02	0.06
7	0.93	0.01	0.06
8	0.96	0.03	0.04
9	0.95	0	0.05
10	0.11	0.78	0.11
11	0.21	0.49	0.3
12	0.23	0.53	0.24
13	0.24	0.54	0.23
14	0.23	0.53	0.26

Table 1: Probabilities of life support system state at time steps 1-14.

In contrast, we run another modified program where after the astronaut wakes up, he begins normal activity inside, as opposed to high exertion activity outside, with results displayed in Table 2. In this case, the network correctly infers that life support is more likely to be in a normal state.

These results demonstrate loopy logic's ability to learn and reason in uncertain situations. In this case, the uncertainty is which state the life support system is in after life support emissions has stopped.

```
person_activity(10)=normal.
person_activity(11)=normal.
person_activity(12)=normal.
person_activity(13)=normal.
person_activity(14)=normal.
person_location(10)=in.
person_location(11)=in.
person_location(12)=in.
person_location(13)=in.
person_location(14)=in.
```

Time	Normal	Stressed	Critical
10	0.96	0	0.04
11	0.69	0	0.31
12	0.76	0	0.24
13	0.76	0	0.24
14	0.76	0	0.24

Table 2: States of life support system when `person_activity` is kept at `normal` and `person_location` is kept at `in`.

To summarize, EM learning takes the form of parameter fitting. A distribution can be used to estimate the value of the learnable parameter. Using our DBAYES algorithm, learning can also be applied to conditional probability tables, not just to variables with simple prior distributions. Learnable distributions can be parameterized with variables just as any other logic term.

In the AR-HMM, we learn the transition probabilities between the 3 states: safe, unsafe and faulted. This distribution may not be known at the beginning of experimental testing. Hence, we can model this distribution as a learnable distribution in which we approximate the transition probability by observing a large set of the training data.

A more complete specification of the OCAML based representation for learning and the loopy belief propagation inference system may be found in Pless and Luger (2001, 2003).

6. Summary and Conclusions

We have created a logic-based stochastic modeling language that has the capability to handle complex situations with repetitive structure. Since the language is recursive, it is possible to build and analyze models that are represented by a potentially infinite set of databases. The US Navy has provided us with sensor data from helicopter rotor systems that have this property. Modeling potentially infinite databases means that we can efficiently represent time-series processes and various different forms of Markov models.

A well-known and effective inference algorithm, loopy belief propagation (Pearl, 1988), supports inference in our language. Within this first-order logic-based stochastic language the combination rule for complex goal support is

the product distribution. Finally, a form of EM parameter learning is supported naturally within this looping inference framework. From a larger perspective, each type of logic (deductive, abductive, and inductive) can be mapped to elements of our declarative stochastic language: The ability to represent rules and chains of rules is equivalent to deductive reasoning. Probabilistic inference, particularly from symptoms to causes, represents an example of abductive inference, and learning through fitting parameters to known data sets, is a form of induction.

In this paper we demonstrated a actual application of fault diagnosis in complex mechanical systems. We have modeled raw time series data as an AR-HMM. We used recursion within our inference scheme to represent the AR-HMM as well to infer and calculate the transition probabilities between states. We used this knowledge to infer the probability of future faults. We achieved a high accuracy in this process. We also demonstrated how Loopy Logic can perform learning in the context of the AR-HMM. Thus, this application demonstrates the power of a first-order stochastic system to represent and reason with complex models and potentially infinite time-series data.

An ongoing effort in this research is to integrate into the language the semantics of making calls to external computing tools like MATLAB or other library utilities by providing syntactical support in the language itself. When dealing with complex and intractable data formats, like time series data and RGB images, it becomes cumbersome to perform mathematical transforms or computations using the first-order system itself. At these junctures, we find it useful to outsource this job to an off-the-shelf system like MATLAB or some other suitable library for operations like correlation, data format translation, and normalization. The first-order system can deal well with discrete or multinomial data but is not suited to deal with real valued or non-discrete data. The call and return of such external computation should be seamless and somewhat transparent to the modeler.

Another direction for developing our stochastic modeling language is to extend it to include continuous random variables. We also plan to extend learning from parameter fitting to full model induction. Getoor et al. (2001) and Segal et al. (2001) consider model induction in the context of more traditional Bayesian Belief Networks and Angelopoulos and Cussens (2001) and Cussens (2001) in the area of Constraint Logic Programming. Finally, the Inductive Logic Programming community (Muggleton, 1994) also addressed the learning of structure with declarative stochastic representations. We plan on taking a combination of these approaches.

Acknowledgments

The research supporting the original development of our stochastic modeling language was provided by NSF (115-9800929, INT-9900485). The follow-up development of a Java-based tool kit for building stochastic models, DBAYES, addressing problems for the US Navy, was supported by a NAVAIR SBIR (N00T001) and STTR (N0421-03-C-0041). We thank Carl Stern and Management Sciences, Inc of Albuquerque, New Mexico for their help in this research and development. We also thank Bill Hardman of Navair Patuxent River Naval Air Station for introducing us to his helicopter research facility. Finally, the development of the original OCAML version of DBAYES was a component of Dan Pless' PhD research in the Computer Science Department at the University of New Mexico.

References

Angelopoulos, N., and Cussens, J. 2001. Markov Chain Monte Carlo Using Tree-Based Priors on Model Structure. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, San Francisco.: Morgan Kaufmann.

Angelopoulos N., and Cussens J., Prolog issues of an MCMC algorithm. In *Proceedings of the 14th International Conference of Applications of Prolog INAP2001*, pages 246-253, Tokyo, Japan, October 2001.

Chakrabarti, C. 2005. *First-Order Stochastic Systems for Diagnosis and Prognosis*, Masters Thesis, Dept. of Computer Science, University of New Mexico.

Cussens, J. 2001. Parameter Estimation in Stochastic Logic Programs, *Machine Learning* 44:245-271.

Dagum, P., Galper, A., and Horowitz, E. 1992. Dynamic Network Models for Forecasting. In *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, 41-48. Morgan Kaufmann.

Dempster, A.P., Laird, N.M., and Rubin, D.B. (1997). Maximum Likelihood from Incomplete Data Via the EM Algorithm. *J. of the Royal Stat. Soc.* Series B, 39, 1, 1-38.

Getoor, L., Friedman, N., Koller, D., and Pfeffer, A. 2001. Learning Probabilistic Relational Models. *Relational Data Mining*, S. Dzeroski and N. Lavorac (eds).: Springer.

Juang, B. 1984. On the Hidden Markov Model and Dynamic Time Warping for Speech Recognition: a unified view, Technical Report, vol 63, 1213-1243 AT&T Labs.

Kersting, K. and De Raedt, L. 2000. Bayesian Logic Programs. In *AAAI-2000 Workshop on Learning Statistical Models from Relational Data.* Menlo Park, CA.: AAAI Press.

Koller, D., and Pfeffer, A. 1998. Probabilistic Frame-Based Systems. In *Proceedings of the Fifteenth National*

Conference on AI, 580-587. Cambridge, MA.: MIT Press.

Mayraz, G., and Hinton, G. 2000. Recognizing Hand-Written Digits using Hierarchical Products of Experts. *Advances in Neural Information Processing Systems 13:* 953-959, 2000.

Muggleton, S. 1994. Bayesian Inductive Logic Programming. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory,* 3-11. New York.: ACM Press.

Ng, R. and Subrahmanian, V. 1992. Probabilistic Logic Programming. *Information and Computation*: 101-102.

Ngo, L., and Haddawy, P. Answering Queries from Context-Sensitive Knowledge Bases. *Theoretical Computer Science* 171:147-177, 1997.

Pearl, P. 1988. Probabilistic Reasoning in Intelligent Systems: *Networks of Plausible Inference.* San Francisco CA.: Morgan Kaufmann.

Pless, D., and Luger, G.F. 2001. Toward General Analysis of Recursive Probability Models. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, San Francisco.: Morgan Kaufmann.

Pless, D., and Luger, G.F. 2003. EM Learning of Product Distributions in a First-Order stochastic Logic Language. *IASTED Conference*, Zurich.: IASTED/ ACTA Press.

Segal, E., Koller, D. and Ormoneit, D. 2001. Probabilistic Abstraction Hierarchies. *Neural Information Processing Systems,* Cambridge, MA.: MIT Press.